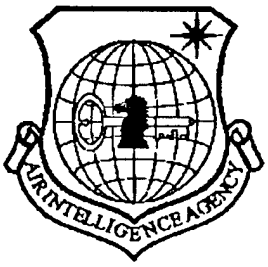
NAIC-ID(RS)T-0021-95

NATIONAL AIR INTELLIGENCE CENTER



INTERMEDIATE LANGUAGE DESIGN AND REALIZATION ASSOCIATED WITH PC DECOMPILATION SYSTEMS

by

Li Hongmang, Liu Zongtian, Zhu Yifen

Approved for public release: distribution unlimited

HUMAN TRANSLATION

NAIC-ID(RS)T-0021-95 26 May 1995

MICROFICHE NR: 95C000327

INTERMEDIATE LANGUAGE DESIGN AND REALIZATION ASSOCIATED WITH PC DECOMPILATION SYSTEMS

By: Li Hongmang, Liu Zongtian, Zhu Yifen

English pages: 17

Source: Xiaoxing Weixing Jishanji, Vol. 12, Nr. 2, 1991; pp. 23-28; 46

Country of origin: China
Translated by: SCITRAN
F33657-84-D-0165
Requester: NAIC/TATA/Keith D. Anthony
Approved for public release: distribution unlimited.

GRAPHICS DISCLAIMER

All figures, graphics, tables, equations, etc. merged into this translation were extracted from the best quality copy available.

INTERMEDIATE LANGUAGE DESIGN AND REALIZATION ASSOCIATED WITH PC DECOMPILATION SYSTEMS

Li Hongmang Liu Zongtian Zhu Yifen

ABSTRACT

Intermediate language, in decompilation systems, takes on an intermediary function associated with a link connecting what comes first with what comes after. Since it is a more advanced standardization of compilation language, it also contains basic structures associated with high level languages. This article introduces intermediate language design and realization methods associated with C language decompilation systems on PC machines.

I. INTRODUCTION

Decompilation acts as a software understanding tool and is the key constituent part associated with reversed software processes. The function is to take machine code and transform it into functionally equivalent high level language which is easy to understand and protect. During transformation processes, first of all, structural analysis and functional analysis is carried out on machine code. Following that, on the basis of appropriate matching transformation rules, program control flow paths are taken and transformed from linear structures into structured forms. At the same time, variables in machine code are taken and, utilizing information recovery, restored to become higher level language variable and type explanations.

On Dual 68000, we already realized decompilation systems from 68000 to C [1]. In order to make decompilation techniques possess even more adaptable and practical natures, at the present time, we carried out design and realization of decompilation systems on PC machines. Due to PC machine command flexibility and symbol table information not being included in target codes, as a result, degrees of difficulty and complexity were relatively

great. In order to lower system complexity, modualized structures associated with systems were strengthened. We opted for the use of decompilation methods associated with using intermediate language as intermediary. First of all, 8086/8087 commands were taken and transformed into intermediate language. After that, on intermediate language boundary surfaces, control flow and data flow analyses were carried out. Due to unique characteristics of PC machines, intermediate language design requires confronting flexible 8086 command systems. As a result, degrees of difficulty are relatively great. However, with regard to the adaptability and practical nature of decompilation technology, it will play important roles.

II. DESIGN THINKING ASSOCIATED WITH INTERMEDIATE LANGUAGE

The generality of intermediate languages are mostly used in situations geared to the needs of machine language transformations, that is, transformations are from high level forms into intermediate code. After that, it is reformed into machine code. If, in compilation programs, use is made of such intermediate codes as ternary forms and quaternary forms, they will often stress compatibility downward. As a result, such problems as code efficiencies, capability of transplantation, and memory space occupied by code are inevitably encountered. However, during decompilation, intermediate language is geared toward high level language transformations. It fits between machine language and high level language. Since it is a further standardization of machine language, it necessarily also contains basic structures associated with high level language. As a result, intermediate language requirements in decompilation are also even more complicated. If it will be able to display fixed categories of machine language, it will also possess certain independent characteristics. The representation forms and structures must also be appropriate to program analysis,

restructuring, and transformation. Because of this, a practical intermediate language geared to high level language transformations possesses the characteristics below:

(1) All operation numbers opt for the use of unified forms of processing.

(2) All operation numbers are uniformly capable of clarification inquiries.

(3) Space occupied by commands and data are not mutually related, that is, the two do not occupy the same address space.

(4) Command block physical sequences are independent of the physical sequences associated with original machine language programs. /24

(5) Command and operation number forms must be appropriate to data type decisions.

(6) Intermediate languages should possess certain "deformation" capabilities, that is, possess a number of new organizational control flow path commands in order to suit the level by level abstraction associated with control analysis.

(7) Intermediate language program structures should opt for the use of similar high level language program structures in order to fit the production of high level language programs.

III. INTERMEDIATE LANGUAGE DESIGN

In decompilation processes, intermediate languages are intermediaries between machine language and high level language. Key processes associated with decompilation are all carried out in this intermediate language. As a result, the design of practical intermediate languages is unusually important [2]. During the development processes of this task, on the basis of the characteristics of 8086 machine language and C language, we meticulously designed an intermediate language which is relatively comprehensive. It primarily contains six constituent parts: intermediate language program structures, command systems, addressing forms, command block types, data types, and control flow graphics. Below we discuss each one in detail.

1. Intermediate Language Program Structures

Among high level languages, some are layered structures--for example, such languages as PASCAL and ALGOL. Some are nonlayered parallel structures--for example, function type language C and so on. Whether or not they are layered or nonlayered, target codes formed after high level language compilation are uniformly linear structures. The relationships of the various subprograms are all contained among transfer relationships associated with functions and processes. Due to the disappearance of such high level language components as explain forms of expression and data types, limits associated with functions and processes in machine code have all already disappeared. Because of this, intermediate language program structures we defined are function type structures. Functions can have reentry values. They also may not have reentry values. Functions and function relationships are parallel. They can be inserted in sets and transferred. However, it is not permitted to insert definitions in sets. Functions are the execution unit of the intermediate languages in question. On the basis of function names, it is possible to define a specially designated name main( ) to act as main function. Within the main function, it is possible to transfer any other function.

Intermediate language program = <main function>

{<function>}m m=0,1,...

<function> = <function head> <function body> <function tail>

<function> = PROGRAM <function name>; <function no.>;

<original machine language address>

<function body> = {<command block>}n n=0,1,2 ...

<function tail> = END

2. Intermediate Language Command Systems

First of all, we discuss a bit one type of abstraction device needed by intermediate language. The characteristics are as follows:

(1) Adequately large memory storage devices.

(2) Adequately numerous data registers and address registers.

(3) Intermediate language statement forms are quaternary forms, that is:

<operation code> <operation no.1>; <operation no.2>;

<operation no.3>

This type of form is similar to quaternary form intermediate code utilized in compilation systems. The advantages are:

A: Forms and compilation language command forms approach each other. It is easy to take various types of compilation commands and reflect them in the intermediate language in question.

B: Quaternary forms have very strong flexibility. They are advantageous to the production of high level language expression forms.

(4) Operation numbers are composed of three parts: addressing forms, variable names, displacement amounts.

(5) Possesses general arithmetical operations, logic operations, relational operations, control flow operations, and other such commands.

(6) Has function interior restructure command. /25

From the characteristics of abstract devices discussed

above, intermediate language commands which we designed can be divided into the several categories below:

(1) function commands (2) control commands (3) restructuring commands.

3. Intermediate Language Addressing Forms

As far as command systems associated with various types of machines are concerned, the individual addressing forms are different. In conjunction with this, they are dependent on the individual machines. In view of the functions of intermediate language in decompilation systems, when designing intermediate language addressing forms, we should, as much as possible, eliminate unique characteristics associated with command addressing of the various types of machines. In conjunction with that, we should abstract and standardize them. Despite various types of machine command addressing forms each possessing special characteristics, they, however, also have a common character. With regard to the addressing of any operation number, in general, they are included in nothing more than immediate numbers, data or addresses taken from registers, and data or addresses taken from memory storage units. Because of this, the intermediate language addressing forms which we designed are:

(1) Immediate Addressing (IA)

(2) Direct Addressing (Register Direct Addressing or Memory Storage Unit Direct Addressing) (DA)

(3) Indirect Addressing (Register Indirect Addressing or Memory Storage Unit Indirect Addressing) (IDA)

4. Intermediate Language Command Block Types

Target code formed after translation of high level language programs--through decompilation--forms into compilation programs. Moreover, in general compilation language text, before each subprogram, there are several related commands and functions introduced. However, these do not correspond to the function body statements of the original program. During decompilation processes, there is only identification of the sections in

question, and semantic transformations are not carried out. As a result, we designate these commands as subprogram head null commands. By the same reasoning, subprogram end sections also have several commands related to escape functions. These are designated subprogram end null commands. Subprogram head null commands and subprogram end null commands are all designated as null commands.

• Valid Commands: The commands between subprogram end null commands and subprogram head null commands all correspond to function body statements. These commands are designated as valid commands.

• Block Header Lines: (1) the first valid command of a function (or subprogram); (2) directional commands associated with direct or conditional transfer commands; (3) that command following direct or conditional transfer commands.

• HALT Block: A blank command block placed in a null command position associated with function or subprogram end sections (it only contains one blank command line) is designated as HALT block. In conjunction with this, it is stipulated that all valid commands associated with directional subprogram end null commands are transferred toward HALT block header lines.

• Start Block: In front of each valid command associated with functions (or subprograms), one start block is defined. It marks the beginning of functions.

• HALT Block Complex: (1) HALT blocks belong to HALT block complexes; (2) if block i contains only one direct transfer command and is transferred toward a certain block belonging to a HALT block complex, then, block i also belongs to the HALT block complex.

• Valid Command Blocks: In accordance with written functions, commands from one block header line to in front of the next block header line belong to the same valid block.

• Forerunner Blocks and Follow On Blocks: if the final block EB_i command is a direct transfer command or conditional transfer command, and it is directed toward block EB_j or the final EB_i command is not a transfer command, however, EB_j uses written functions to follow right behind EB_i, then, EB_i is designated as a forerunner block of EB_j. EB_j is a follow on block of EB_i.

• ED Block Complex: if EB_i block does not belong to the HALT block complex, however, it at least has a follow on block that belongs to the HALT block complex, then, EB_i is designated as belonging to ED block complex.

Note: The primary purpose of defining ED block complex is for the sake of solving for subprogram (or function) reentry values.

On the basis of intermediate language command classifications and characteristics as well as definitions of previously discussed block concepts, we take intermediate language command blocks and divide them into the several types below.

(1) Start Block: as previously defined.

(2) Function Block: only has one outcome.

(3) Decision Block: has two outcomes.

(4) Multi-branch Block: has multiple outcomes.

(5) HALT Block: as previously defined.

As far as the five types of blocks above are concerned, in intermediate language control graphics, they form into five basic control flow graphic nodes.

5. Intermediate Language Data Types

High level languages utilize rich data structures. In conjunction with this, through data types, statements are explained in order to explain definitions. However, these statements have already not come back into existence after going through compilation. In target code, there are also no corresponding code sections. On the basis of compilation principles, it is possible to know that compilation programs--in accordance with high level language variables--explain data type tables associated with statement structure variables. In the tables are recorded such information as the nomenclature of relevant variables, distribution addresses, and so on. During code formation, high level language variables are taken and reflected in the address elements or registers distributed to. In conjunction with this, on the basis of variable types and operation natures, corresponding commands and operation numbers are produced.

Command systems associated with various types of machines all contain rich addressing forms. When high level language is

realized on the machines in question, inspections are carried out of different high level language data types in order to precisely determine appropriate addressing forms so as to satisfy the recovery from memory of various types of variables. Because of this, high level language variable data type information in machine language certainly does not completely disappear. It is contained hidden in the following areas:

(1) operation codes (2) operation numbers and address forms (3) distribution characteristics associated with various registers.

In decompilation processes, we must go through data type analysis, taking this implicitly contained information and recovering it in full. In conjunction with this, level by level abstract induction forms high level language data types.

Intermediate language--during decompilation processes--takes on the function of forming a link between what precedes it and what follows it. As a result, intermediate language should be able to completely inherit data variable information in target code. In conjunction with this, to a certain degree, abstract induction forms basic data types. Following that, after data type analysis, abstraction forms data types similar to ones in high level language.

Intermediate language basic data types are: T_i (integers), length is 1, 2, 4 character segments; T_u (numbers without symbols), length is 2 character segments; T_f (floating point numbers), length is 4, 8 character segments; T_b (Boolean numbers).

Intermediate language composite types are: POI, FUN, ARRAY, Union, MARRAY, Struc.

On the basis of differing functional areas associated with various types of intermediate language variables, it is possible to take intermediate language variables and divide them into overall variables (L_k), form parameter variables (IP_k), local variables (LV_k), immediate numbers (IN), index register variables (RB_k), data register variables (D_k), temporary index register

variables (TR_k), temporary data register variables (TD_k), and supplementary local variables (T_k). Among these, T_k is for the sake of facilitating the production of forms of expression. Besides that, there are a number of temporary variables introduced. TD_k and TR_k are temporary variables used within systems. They do not correspond to any variables in high level language programs.

6. Intermediate Language CFG (Control Flow Graphic)

Above, we defined intermediate language blocks and their types. Here, we introduce several basic concepts besides these in order to describe the relationships between them, thereby describing control structures associated with intermediate language programs.

• Definition 1: Program control flow graphics are single flow graphs G=(V, E, n, f). Among these: 1) in flow graphics, each node $V_i \in V$ expresses one block bi in programs; 2) in flow graphics, each directional side $c_{ij} = \langle V_i , V_j \rangle \in E$ corresponds to the relationship between blocks bi and bj; 3) assuming nodes $V_i , V_j \in V$ correspond to blocks bi and bj, then, the condition which exists on side $c_{ij} = \langle V_i , V_j \rangle \in E$ is that bi is a precursor block of bj; 4) f corresponds to HALT block.

• Definition 2: Transfer graphics are single flow graphs $G=(V, E, n_0, f)$. Among these: 1) V represents various subprograms within programs. $n_0 \in V$ is the main program; 2) E represents the assembly of all sides in graphics. Directional side $c_{ij} = \langle V_i , V_j \rangle \in E$ represents subprogram V_i single or multiple iterations of transfer associated with subprogram V_i; 3) f represents the most basic subprogram. It does not transfer any other subprogram.

Decompilation is a transformation process associated with functional equivalence. Speaking in terms of control structures, if one wishes to guarantee the equivalence, then, it is necessary to precisely preserve corresponding execution function equivalences between original machine language code and high level language programs produced by decompilation, that is, dynamic execution functions are the same. Due to this type of relationship, after the formation of intermediate language from machine language, intermediate language does not then need to be concerned with the physical placement of various execution code sections. The only concern is for program execution functions. Moreover, program execution functions are determined by its control flow direction. As a result, in the process of designing intermediate language, we opted for the use of control flow graphic CFG unrelated to machine code physical placement in order to describe program execution functions, thereby guaranteeing the equivalence of the two. In conjunction with this, it is possible to make control flow analysis relatively independent (only

carried out on CFG). CFG has five types of nodes, that is, SN (Start Node), FN (Function Node), DN (Decision Node), TN (Terminal Node), and MN (Multi-branch Node). In conjunction with this, they respectively correspond to the five types of blocks.

IV. INTERMEDIATE LANGUAGE REALIZATION

In the process of realizing decompilation, intermediate language takes on the role of forming a link between what precedes it and what comes after it. On the one hand, it requires inheriting all information associated with compilation language. On the other hand, it also must carry out appropriate abstract induction on compilation language. Because of this, in the process of realizing intermediate language, there are two primary parts included: intermediate language program production and intermediate language variable information production. The former--on the basis of functional equivalence--takes compilation commands and transforms them into corresponding intermediate language commands. The latter--on the basis of utilization information associated with data variables in compilation language--goes through appropriate recovery and transformation methods and produces corresponding variable information at relatively high levels.

1. Intermediate Language Program Production

On PC machines, turning compilation language into intermediate language programs can be realized through the processes described below:

(1) Scanning compilation language programs, on the basis of control flow, take compilation programs and divide then into block order.

(2) On the basis of relationships between various blocks, construct compilation language program control flow graphics.

(3) Using blocks as units, take compilation language and transform it into equivalent intermediate language commands.

(4) Form intermediate language control flow graphics CFG.

When transforming compilation language into equivalent intermediate language commands, it is necessary to pay attention to the several points below:

(a) As much as possible, eliminate redundant commands in compilation language and do not lose various types of information.

(b) Identify macro transfers and internal storage functions. In conjunction with this, obtain their parameters.

(c) Identify 8087 command operation numbers and their functions.

(d) With regard to conditional transfer commands, on the basis of conditional code attributes, change them into corresponding comparative commands and true false transfer (GOT/GOF) commands.

Intermediate language control flow graphics are a weighted directional graphic. Each node corresponds to a block. The relationships between various nodes are described through weighted values as shown in Fig.1. In this, T: represents a true outcome. F: represents a false outcome. M: represents a multi-branch outcome.

2. Formation of Simple Symbol Tables /28

Executable documents formed after software compiled in high level language goes through compilation generally do not contain symbol table information. Furthermore, there are no clear variable type explanations. In them, only utilization information associated with variables is contained. As far as the formation of simple symbol tables is concerned, they are simple properties associated with variables and recovered by induction on the basis of variable utilization information.

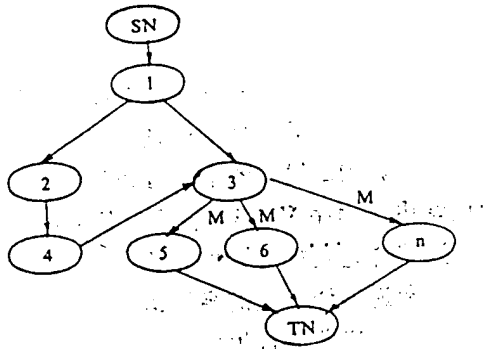


Fig.1 Intermediate Language Control Flow Graphic Sample

Variable utilization information primarily contains: variable distribution rules, operation number length and addressing methods, operation code semantics, function form parameter and real parameter transmission and utilization, and so on. When scanning and analyzing each function command, variable properties are taken one by one and entered in corresponding tables thereby constructing the simple symbol table shown in Fig.2. In it, TN: type nodes are used to describe variable and function types.

V. CONCLUDING REMARKS

Intermediate language design and realization are the main parts in decompilation processes. Intermediate language control flow paths and variable information are still a type of relatively low level form. It is still necessary during later control flow analysis and data type analysis to reconstruct, step by step, from lower level forms, high level language forms of control flow paths and data types.

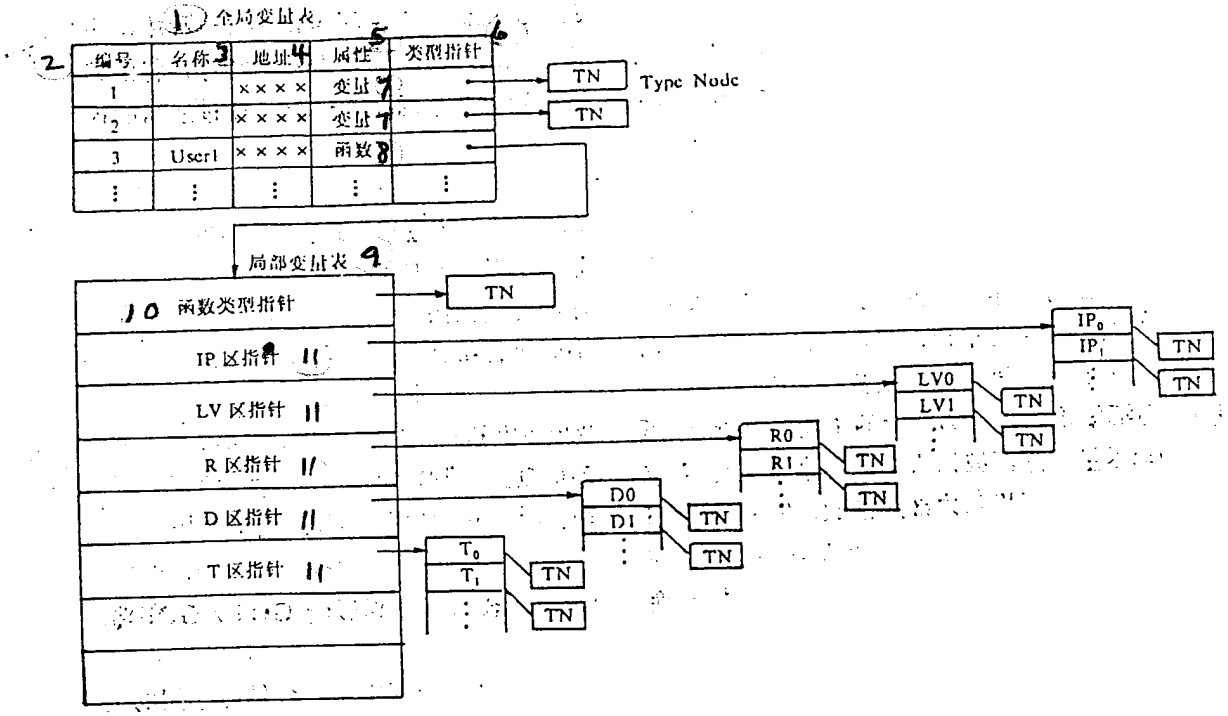


Fig.2 Simple Symbol Table (1) Overall Variable Table (2) Serial No. (3) Nomenclature (4) Address (5) Attribute (6) Type POI (7) Variable (8) Function (9) Local Variable Table (10) Function Type POI (11) Zone POI

REFERENCES

1) 刘宗田、朱逸芬，符号执行技术在68000C反编译系统中的应用，计算机学报，11:10(1988)633～637.
2) 朱逸芬、史展和、申利民、曹蕴锐、李宏芒，基于中间语言的逆编译系统CDS，合肥工业大学学报，11:4(1988)1～7.
3) 刘宗田，68000C反编译程序中的语句翻译器的设计与实现，小型微型计算机系统，9:2(1988)1～10.

DISTRIBUTION LIST

DISTRIBUTION DIRECT TO RECIPIENT

ORGANIZATION	MICROFICHE
B085 DIA/RTS-2FI	1
C509 BALLOC509 BALLISTIC RES LAB	1
C510 R&T LABS/AVEADCOM	1
C513 ARRADCOM	1
C535 AVRADCOM/TSARCOM	1
C539 TRASANA	1
Q592 FSTC	4
Q619 MSIC REDSTONE	1
Q008 NTIC	1
Q043 AFMIC-IS	1
E404 AEDC/DOF	1
E408 AFWL	1
E410 AFDTC/IN	1
E429 SD/IND	1
P005 DOE/ISA/DDI	1
P050 CIA/OCR/ADD/SD	2
1051 AFIT/LDE	1
PO90 NSA/CDB	1

Microfiche Nbr: FTD95C000327L
NAIC-ID(RS)T-0021-95